Apuntes U 3

Tema ***Divide y vencerás, Recursividad, aritmética Modular, MCD, Criptografía***

"Los sabios buscan la sabiduría; los necios creen haberla encontrado"
Napoleón

Divide y vencerás

Puede considerarse como una filosofía para resolver problemas. Es una técnica para el diseño de algoritmos que consiste en resolver un problema a partir de la solución de sub-problemas del mismo tipo, pero de menor tamaño.
Si los sub-problemas son todavía relativamente grandes se aplicará de nuevo esta técnica hasta alcanzar sub-problemas lo suficientemente pequeños para ser solucionados directamente.

Pasos para aplicar la técnica de "divide y vencerás":

1- Plantear el problema de forma que pueda ser descompuesto en k sub-problemas del mismo tipo, pero de menor tamaño. Es decir, si el tamaño de la entrada es n, hemos de conseguir dividir el problema en k sub-problemas (donde $1 \leq k \leq n$), cada uno con una entrada de tamaño nk y donde $0 \leq nk < n$. A esta tarea se le conoce como división.
   El número k debe ser pequeño e independiente de la entrada determinada, por ejemplo: en el caso particular de contener una sola llamada recursiva tendremos $k = 1$
   Donde llegamos a este tipo de algoritmo, podemos hablar de simplificación; ejemplos: el cálculo de la factorial de un número. También son algoritmos de simplificación el de búsqueda binaria en un vector o el que resuelve el problema del k-ésimo elemento.
2- En segundo lugar han de resolverse independientemente todos los sub-problemas, bien directamente si son elementales o bien de forma recursiva. El hecho de que el tamaño de los sub-problemas sea estrictamente menor que el tamaño original del problema nos garantiza la convergencia hacia los casos elementales, también denominados casos base.
3- Por último, combinar las soluciones obtenidas en el paso anterior para construir la solución del problema original.

¿Que NO es la recursividad?

Primero debemos decir que la recursividad no es una estructura de datos, sino que es una técnica de programación que nos permite que un bloque de instrucciones se ejecute **n** veces. Remplaza en ocasiones a estructuras **repetitivas o iterativas**.

¿Qué es?

Técnica de programación muy potente que puede ser usada en lugar de la iteración.

Algoritmo recursivo

Un algoritmo recursivo es un algoritmo que expresa la solución de un problema en términos de una **llamada a sí mismo**. La llamada a sí mismo se conoce como llamada recursiva o recurrente.

```
FUNCIÓN Factorial(n)
    VAR resultado: Entero

    SI (n<2) ENTONCES
        resultado = 1;
    SINO
        resultado = n * Factorial(n-1);
    FSI

    RETORNA resultado;
FUNCIÓN
```

La recursividad es uno de los conceptos más importantes en programación y es una capacidad que tienen los sub-programas a auto invocarse.
La recursividad nos permite trabajar donde las estructuras del tipo iterativas vistas en la Unidad 1 no son de fácil aplicación. Esto nos indica que **no hay problemas intrínsecamente recursivos o iterativos**; cualquier proceso puede expresarse de una u otra forma.

Si bien la recursividad, como veremos, da lugar a algoritmos más fáciles de seguir y entender, debemos tener en cuenta que consume más recursos y es más lenta que una solución iterativa.
Para desarrollar algoritmos recursivos hay que partir del supuesto de que ya existe un algoritmo que resuelve una versión más sencilla del problema.
A partir de esta suposición debe hacerse lo siguiente:

1. Identificar sub-problemas atómicos de resolución inmediata (casos base).
2. Descomponer el problema en sub-problemas resolubles mediante el algoritmo pre-existente; la solución de estos sub-problemas debe aproximarnos a los casos base.
3. Probar de manera informal que tanto los casos base como los generales pueden solucionarse con el algoritmo desarrollado.

Se pueden presentar uno o varios, nuevamente dependiendo del problema a resolver.
Cuando se analiza la solución recursiva de un problema es importante determinar con precisión cuáles serán los pasos básico y recursivo.
En cada vuelta del ciclo es importante que nos acerquemos cada vez más a la solución del problema en cuestión, es decir, al paso básico. Si esto no ocurre podemos estar en un ciclo extraño.
Puede que el problema se encuentre mal definido y entonces la máquina se quedaría ejecutando por tiempo indefinido el programa y sólo terminaría al agotarse la memoria.
Como todas las funciones recursivas tienen la misma estructura, el cuerpo de la función será un condicional.
¿Cuántas condiciones debo poner?

- Una por cada caso base
- Una por el caso recursivo
Se debe probar primero el caso base, porque si éste tiene errores (lógicos) es posible que la función se quede haciendo un ciclo infinito.

Tipos de recursividad

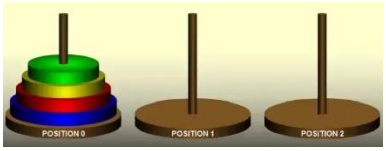
- **Directa:** el programa o subprograma se llama directamente a si mismo.
- **Indirecta:** el programa o subprograma llama a otro y éste en algún momento llama al primero.

Claves de la recursividad

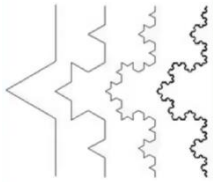
Los compiladores y/o intérpretes deben tener características de aceptar recursividad (en la actualidad todos o la gran mayoría lo tienen).
La recursividad nos permite trabajar donde las estructuras del tipo iterativas vistas en la no son de fácil aplicación. Esto nos indica que no hay problemas intrínsecamente recursivos o iterativos; cualquier proceso puede expresarse de una u otra forma.

Problemas típicos de recursividad

- **Torres de Hanói**



- **Fractales**



- **Sumatorias**

$$S = \sum_{i=0}^{N} a_i$$

- **Sucesiones de Fibonacci**

- **Factorial**

Proceso de recursividad con el cálculo factorial

Supongamos que calcularemos el factorial de 4 (4!).
Sabemos matemáticamente que

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

También que

$$3! = 3 \times 2 \times 1$$

y que

$$2! = 2 \times 1$$

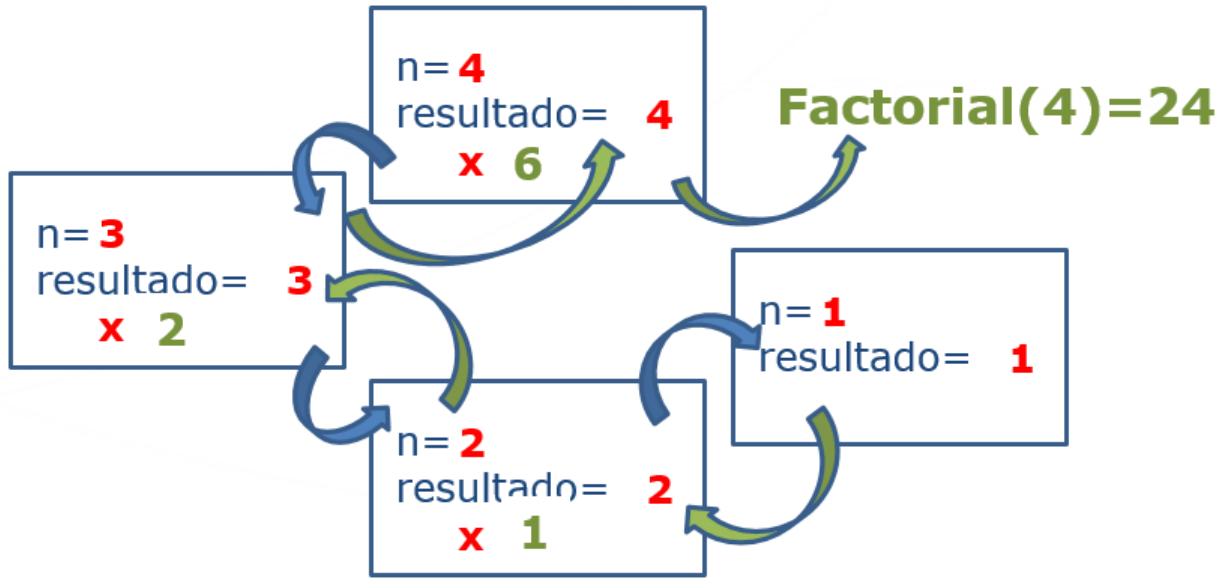
Y por definición

$$1! = 1$$

¿Podemos encontrar un patrón?

Entonces también podemos expresarlo de la siguiente forma:

$$4! = 4 \times 3!$$
$$3! = 3 \times 2!$$
$$2! = 2 \times 1!$$
$$1! = 1$$



Ejemplos de código de ña función factorial recursiva

```
FUNCIÓN Factorial(n)
    VAR resultado: Entero

    SI (n<2) ENTONCES
        resultado = 1;
    SINO
        resultado = n * Factorial(n-1);
    FSI

    RETORNA resultado;
FUNCIÓN
```

```java
package recursividad;

public class factorialrecursivo {
    public int CalcularFactorial (int numero){
        if  (numero <2){
            return 1;
        } else {
            return numero * CalcularFactorial (numero-1);
        }

    }

}
```

Sucesión de Fibonacci (recursiva)

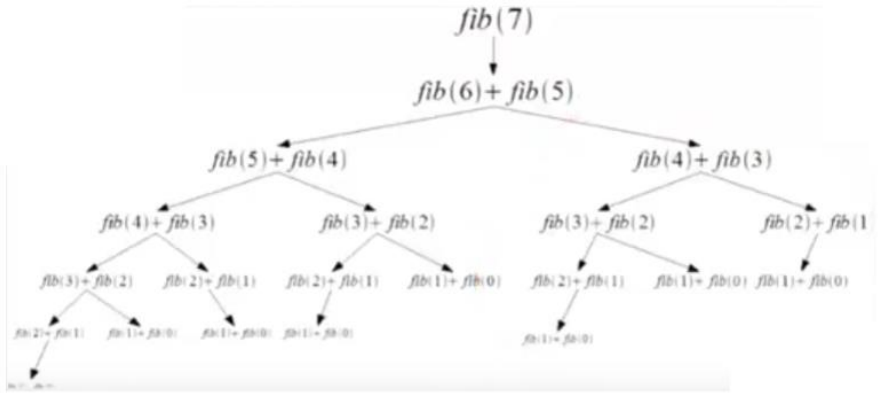
$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597 \ldots$$

La sucesión comienza con los números 0 y 1, y a partir de estos, «cada término es la suma de los dos anteriores», es la relación de recurrencia que la define.
A los elementos de esta sucesión se les llama números de Fibonacci. Esta sucesión fue descrita en Europa por Leonardo de Pisa, matemático italiano del siglo XIII también conocido como Fibonacci. Tiene numerosas aplicaciones en ciencias de la computación, matemática y teoría de juegos. También aparece en configuraciones biológicas, como por ejemplo en las ramas de los árboles, en la disposición de las hojas en el tallo, en las flores de alcachofas y girasoles, en las inflorescencias del brécol romanesco y en la configuración de las piñas de las coníferas. De igual manera, se encuentra en la estructura espiral del caparazón de algunos moluscos, como el nautilus.

Ter	Sucesión(n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

Casos Base

$$f_n = f_{n-1} + f_{n-2}$$



Técnica para implementar recursividad

- Para desarrollar algoritmos recursivos hay que partir del supuesto de que ya existe un algoritmo que resuelve una versión más sencilla del problema..
- A partir de esta suposición debe hacerse lo siguiente:

1. Identificar sub-problemas atómicos de resolución inmediata (***casos base***).
2. Descomponer el problema en sub-problemas resolubles mediante el algoritmo pre-existente; la solución de estos sub-problemas debe aproximarnos a los casos base.
3. Probar de manera informal que tanto los casos base como los generales pueden solucionarse con el algoritmo desarrollado.

Nunca debemos resolver la misma instancia del problema en llamadas recursivas separadas

Demostración por inducción matemática

Se trata de una técnica de demostración que se utiliza para demostrar muchos teoremas que afirman que P(n) es verdadera para todos los enteros positivos n.
En matemáticas, la inducción es un razonamiento que permite demostrar una infinidad de proposiciones, o una proposición que depende de un parámetro n que toma una infinidad de valores enteros. En términos simples, la inducción matemática consiste en el siguiente razonamiento:

* Premisa mayor: El número entero tiene la propiedad P.*

* Premisa menor: El hecho de que cualquier número entero n tenga la propiedad implica que también n+1 la tiene.

Conclusión: Todos los números enteros a partir de a tienen la propiedad P. Con más rigor, el método de inducción matemática es el que realiza la demostración para proposiciones en las que aparece como variable un número natural. Se basa en un axioma denominado principio de la inducción matemática.

Una descripción informal de la inducción matemática puede ser ilustrada por el efecto dominó, donde ocurre una reacción en cadena con una secuencia de piezas de dominó cayendo una detrás de la otra.

PASO BASE: Se muestra que la proposición P(1) es verdadera.

PASO INDUCTIVO: se muestra que la implicación P(k) -> P(k + 1) es verdadera para todo entero positivo k

Complejidad de algoritmos recursivos

Un aspecto interesante de este ejemplo en particular es que la **complejidad** del algoritmo recursivo para el cálculo del factorial es idéntica a la del algoritmo iterativo pues en ambos casos es O(n).

¿Quiere esto decir que ambos algoritmos son igual de eficientes?
No, recordemos que esto es un límite asintótico y prescindiendo de constantes multiplicativas.

Así, ambos algoritmos se comportarán de forma similar: si m es 2n, entonces el cálculo de m! tardará como máximo el doble que n!
Sin embargo, es posible que el algoritmo recursivo tarde más que el iterativo por cuestiones de la implementación de la recursividad en una computadora, no por cuestiones algorítmicas.